The first few readings are intended to provide a high-altitude overview of the questions of design computing, and to introduce one view in some detail. We will refer to this as the “classical theory of problem solving” or “state-space exploration”. They begin with the question of problems…

**Design & Problems**

To establish a theory of design computing it would seem desirable to have a comprehension of both design and computing individually. Let’s begin by asking the question, “What is design?” The disciplines of design are many, including art and architecture as well as, perhaps medicine, law, and music (according to Donald Schön).

Generally speaking, design is defined as a non-obvious plan of action intended to change the environment in such a way as to produce an improvement. The “environment” need not be physical. Doctors *design* treatments. Programmers *design* programs. Some folks *design* experience. However, prescribing aspirin for a headache seems too obvious to be a design. Writing a program to calculate the area of a rectangle hardly requires inspiration. Even selecting a font for a poster may not seem like design. So, the definition of “design” needs to distinguish design from selection among obvious choices, or *rote* (memorized) solution of routine problems, as well as distinguishing design from whim or random choice. Fortunately, we don’t have to accept a fixed narrow definition, just be aware of the range.

Now, to most of us, a suboptimal environment sounds a lot like a *problem*. There have been many approaches to general *problem-solving*. Most approach it in terms of possible design configurations, or *states*, captured in a description, or *representation*, which can be *evaluated* to tell if the desired state has been reached (i.e., the problem has been solved). Sometimes states are characterized in terms of *constraints*, or candidate solutions created by *generative systems*. All of this pretty much assumes you know what the problem *is* when you start. That’s not always the case.

Another view, sometimes called *problem finding* or *puzzle-making*, focuses on the observation that many problems are *not* well-described when we begin and that the solution and the description often co-evolve from a very loose and vague description to a specific and actionable one.

**Computing**

Computers, which most of us think of in terms of mathematics, are most generally characterized as something like “a machine for manipulating symbols,” which covers mathematics, communications, logic, etc. What’s cool, in part, is that both the symbols (*data*) and the instructions for their manipulation (*programs*) can be stored and processed on the same equipment. This allows the computer to switch from being one kind of machine to another, a characteristic generally referred to as *virtuality*.

**Design Computing**

Design computing, then, seems to be concerned with the symbol-processing of humans and/or machines engaged in a problem-solving task called *design*, and how it might be captured, facilitated, enhanced or even automated by the application of computing. This also suggests considering the ways in which *smart* buildings might assist occupants—through management of services (lights, energy, security, etc)—and reflecting on the social ramifications of the use of computing in the production of building designs.

**Questions to Think About**

Do designers fit current problems to pre-existing patterns (corresponding to *problem solving*), or do they (we?) apply strategies for finding both the problem definition and solution simultaneously (corresponding to *puzzle-making*). If the latter, what is the role of *experience*? Where/what is *creativity* in this? What is the role/purpose of drawing?